On Undecidability Results of Real Programming Languages

Wolf Zimmermann
Raimund Kirner Dirk Richter

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

- For real programming languages this argument is not true

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

- For real programming languages this argument could be wrong
- It might be true but in another way than expected

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

- For real programming languages this argument could be wrong
- It might be true but in another way than expected

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

- For real programming languages this argument could be wrong
- It might be true but in another way than expected

Introduction

Frequent Argument for Conservative Program Analysis

Property to be analyzed is undecidable

- live variables
- common sub-expressions
- worst case execution time

Main Argument: Undecidability of the halting problem

This Talk

Critical view on this argument

- For real programming languages this argument could be wrong
- It might be true but in another way than expected

Roadmap

- Consider a representive subset of C (C-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on C-K is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of C (C-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on C-K is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of *C* (*C*-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on *C*-*K* is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of *C* (*C*-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on *C*-*K* is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of C (C-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on C-K is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of C (C-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on C-K is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of *C* (*C*-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on *C*-*K* is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of *C* (*C*-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on *C*-*K* is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of *C* (*C*-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on *C*-*K* is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of C (C-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on C-K is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of *C* (*C*-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on *C*-*K* is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Roadmap

- Consider a representive subset of *C* (*C*-K)
  - Loops, conditionals, assignments, blocks, arithmetic and Boolean expressions
  - local and global variables
  - Only basic datatypes
  - Pointers, heap
  - Procedures and functions (recursion is allowed)
- We show that the semantics can be modeled as a pushdown automaton
- Reachability on pushdown automata is decidable

$\Rightarrow$ Halting problem on *C*-*K* is decidable

☞ Undefined behaviour is modeled

- Difference implementation semantics and language semantics
- Short discussion on other programming languages

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \mathrm{GLOB} \rightharpoonup \mathrm{BIT}^k$
- Heap: $h : \mathrm{BIT}^n \rightharpoonup \mathrm{BIT}^k$
- Only finitely many h

Stack Frames (Stack Alphabet)

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \mathrm{GLOB} \rightarrow \mathrm{BIT}^k$
- Heap: $h : \mathrm{BIT}^n \rightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \text{GLOB} \rightarrow \text{BIT}^k$
- Heap: $h : \text{BIT}^n \rightarrow \text{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \mathrm{GLOB} \rightarrow \mathrm{BIT}^k$
- Heap: $h : \mathrm{BIT}^n \rightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \text{GLOB} \rightarrow \text{BIT}^k$
- Heap: $h : \text{BIT}^n \rightarrow \text{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

- instruction *ip* : LABEL to be executed
- Store for local variables $\sigma_{ip} : \text{LOC}_{ip} \rightarrow \text{BIT}^k$

$\Rightarrow$ Only finitely many stack frames

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \mathrm{GLOB} \longrightarrow \mathrm{BIT}^k$
- Heap: $h : \mathrm{BIT}^n \longrightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

- instruction *ip* : LABEL to be executed
- Store for local variables $\sigma_p : \mathrm{LOC}_p \longrightarrow \mathrm{BIT}^k$
- Registers for storing intermediate values of expressions $\rho : \mathrm{EXPR}_p \longrightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many stack frames

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \text{GLOB} \rightarrow \text{BIT}^k$
- Heap: $h : \text{BIT}^n \rightarrow \text{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

- instruction *ip* : LABEL to be executed
- Store for local variables $\sigma_p : \text{LOC}_p \rightarrow \text{BIT}^k$
- Registers for storing intermediate values of expressions $\rho : \text{EXPR}_p \rightarrow \text{BIT}^k$

$\Rightarrow$ Only finitely many stack frames

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \text{GLOB} \rightarrow \text{BIT}^k$
- Heap: $h : \text{BIT}^n \rightarrow \text{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

- instruction *ip* : LABEL to be executed
- Store for local variables $\sigma_p : \text{LOC}_p \rightarrow \text{BIT}^k$
- Registers for storing intermediate values of expressions $\rho : \text{EXPR}_p \rightarrow \text{BIT}^k$

$\Rightarrow$ Only finitely many stack frames

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \mathrm{GLOB} \rightarrow \mathrm{BIT}^k$
- Heap: $h : \mathrm{BIT}^n \rightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

- instruction *ip* : LABEL to be executed
- Store for local variables $\sigma_p : \mathrm{LOC}_p \rightarrow \mathrm{BIT}^k$
- Registers for storing intermediate values of expressions $\rho : \mathrm{EXPR}_p \rightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many stack frames

Semantics of *C-K* as a pushdown machine

Key observation

All basic data types in *C* are finite

(Global) States

- Store for global variables: $\sigma : \mathrm{GLOB} \rightarrow \mathrm{BIT}^k$
- Heap: $h : \mathrm{BIT}^n \rightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many states

Stack Frames (Stack Alphabet)

- instruction *ip* : LABEL to be executed
- Store for local variables $\sigma_p : \mathrm{LOC}_p \rightarrow \mathrm{BIT}^k$
- Registers for storing intermediate values of expressions $\rho : \mathrm{EXPR}_p \rightarrow \mathrm{BIT}^k$

$\Rightarrow$ Only finitely many stack frames

Semantics of *C-K* as a pushdown machine (C'ted)

Transitions

- Calling a procedure p pushes a stack frame of p onto the runtime stack with the first instruction to be executed
- Assignments etc. change either global store, local store or registers on the top frame
- Returning from a procedure removes the top frame from the runtime stack

The semantics of a program is a pushdown-automaton

Semantics of *C-K* as a pushdown machine (C'ted)

Transitions

- Calling a procedure p pushes a stack frame of p onto the runtime stack with the first instruction to be executed
- Assignments etc. change either global store, local store or registers on the top frame
- Returning from a procedure removes the top frame from the runtime stack

The semantics of a program is a pushdown-automaton


Semantics of *C-K* as a pushdown machine (C'ted)

Transitions

- Calling a procedure p pushes a stack frame of p onto the runtime stack with the first instruction to be executed
- Assignments etc. change either global store, local store or registers on the top frame
- Returning from a procedure removes the top frame from the runtime stack

The semantics of a program is a pushdown-automaton

Semantics of *C-K* as a pushdown machine (C'ted)

Transitions

- Calling a procedure p pushes a stack frame of p onto the runtime stack with the first instruction to be executed
- Assignments etc. change either global store, local store or registers on the top frame
- Returning from a procedure removes the top frame from the runtime stack

The semantics of a program is a pushdown-automaton

Semantics of *C-K* as a pushdown machine (C'ted)

Transitions

- Calling a procedure p pushes a stack frame of p onto the runtime stack with the first instruction to be executed
- Assignments etc. change either global store, local store or registers on the top frame
- Returning from a procedure removes the top frame from the runtime stack

The semantics of a program is a pushdown-automaton

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{n\} \to \mathrm{BIT}^{32}$
- $\Rightarrow 2^{32}$ states

Stack Frames for `main`:

- Local variables: $\sigma : \{x\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$
- $\Rightarrow 6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{n\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$
- $\Rightarrow 13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{n\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow 2^{32}$ states

Stack Frames for `main`:

- Local variables: $\sigma : \{x\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow 6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{n\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow 13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow 2^{32}$ states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow 6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow 13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathsf{BIT}^{32}$
- No heap

$\Rightarrow 2^{32}$ states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathsf{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathsf{BIT}^{32}$

$\Rightarrow 6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathsf{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathsf{BIT}^{32}$

$\Rightarrow 13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow 2^{32}$ states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow 6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow 13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

n=0

0:x=–10	0	2	3
	–7	20	5

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

n=0

0:x=–10	0	2	3
	–7	20	5

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \text{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \text{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \text{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \text{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \text{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \text{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \text{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \text{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \text{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \text{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	15	1	3	7	0	2	5	–1	9	2
3:x=–10	0	2	3							
	2	2	5							

(Global) States:

- $\sigma : \{n\} \to \text{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{x\} \to \text{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \text{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{n\} \to \text{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \text{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	15	1	3	7	0	2	5	–1	9	2
3:x=–10	0	2	3							
	2	2	5							

(Global) States:

- $\sigma : \{n\} \to \text{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{x\} \to \text{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \text{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{n\} \to \text{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \text{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	15	1	3	7	0	2	5	–1	9	2
3:x=–10	0	2	3							
	2	2	5							

n=2

15:n=2	6	7	8	10	12	13	14	15	16	17
	2	1	0	7	2	1	1	–1	9	2
3:x=–10	0	2	3							
	2	2	5							

Example

```
int n;
int fak(int n) {
  if[9] (n[6]<=[8]1[7]) return[11] 1[10];
  return[18] fak[15](n[12]-[14]1[13])*[17]n[16];
}
void main() {
  n=[1]2[0];
  int x=[4]fak[3](n[2]);
  return[5];
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow$ 2^{32} states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow$ $13 \cdot 2^{2048}$ stack frames

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	15	1	3	7	0	2	5	–1	9	2
3:x=–10	0	2	3							
	2	2	5							

n=2

15:n=2	6	7	8	10	12	13	14	15	16	17
	2	1	0	7	2	1	1	–1	9	2
3:x=–10	0	2	3							
	2	2	5							

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow 2^{32}$ states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow 6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow 13 \cdot 2^{2048}$ stack frames

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	15	1	3	7	0	2	5	–1	9	2

3:x=–10	0	2	3
	2	2	5

n=2

15:n=2	6	7	8	10	12	13	14	15	16	17
	2	1	0	7	2	1	1	–1	9	2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	0	9	7	–1	8	27	88	–6	3	1
15:n=2	6	7	8	10	12	13	14	15	16	17
	2	1	0	7	2	1	1	–1	9	2

3:x=–10	0	2	3
	2	2	5

Example

```
int n;
int fak(int n) {
  if (n<=1) return 1;
  return fak(n-1)*n;
}
void main() {
  n=2;
  int x=fak(n);
  return;
}
```

(Global) States:

- $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- No heap

$\Rightarrow 2^{32}$ states

Stack Frames for `main`:

- Local variables: $\sigma : \{\mathtt{x}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{0, \ldots, 5\}$
- Registers: $\rho : \{0, 2, 3\} \to \mathrm{BIT}^{32}$

$\Rightarrow 6 \cdot 2^{128}$ stack frames

Stack Frames for `fak`:

- Local variables: $\sigma : \{\mathtt{n}\} \to \mathrm{BIT}^{32}$
- Program points: $p \in \{6, \ldots, 18\}$
- Registers: $\rho : \{6, 7, 8, 10, 12, \ldots, 17\} \to \mathrm{BIT}^{32}$

$\Rightarrow 13 \cdot 2^{2048}$ stack frames

n=0

0:x=–10	0	2	3
	–7	20	5

n=2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	15	1	3	7	0	2	5	–1	9	2

3:x=–10	0	2	3
	2	2	5

n=2

15:n=2	6	7	8	10	12	13	14	15	16	17
	2	1	0	7	2	1	1	–1	9	2

3:x=–10	0	2	3
	2	2	5

n=2

6:n=2	6	7	8	10	12	13	14	15	16	17
	0	9	7	–1	8	27	88	–6	3	1
15:n=2	6	7	8	10	12	13	14	15	16	17
	2	1	0	7	2	1	1	–1	9	2

3:x=–10	0	2	3
	2	2	5

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

- Integers, Reals etc. are finite
- References are infinite, but it is finite for every implementation

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

- Integers, Reals etc. are finite
- References are infinite, but it is finite for every implementation
- Fundamental difference to *C-K*: reference parameters, local procedures, and procedure parameters that passes closures

$\Rightarrow$ Half-band Turingmachine can be simulated on the runtime stack

$\Rightarrow$ Halting Problem is undecidable

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

- Integers, Reals etc. are finite
- References are infinite, but it is finite for every implementation
- Fundamental difference to *C-K*: reference parameters, local procedures, and procedure parameters that passes closures

☞ Half-band Turingmachine can be simulated on the runtime stack

$\Rightarrow$ Halting Problem is undecidable

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

- Integers, Reals etc. are finite
- References are infinite, but it is finite for every implementation
- Fundamental difference to *C-K*: reference parameters, local procedures, and procedure parameters that passes closures

☞ Half-band Turingmachine can be simulated on the runtime stack

$\Rightarrow$ Halting Problem is undecidable

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

- Integers, Reals etc. are finite
- References are infinite, but it is finite for every implementation
- Fundamental difference to *C-K*: reference parameters, local procedures, and procedure parameters that passes closures

☞ Half-band Turingmachine can be simulated on the runtime stack

$\Rightarrow$ Halting Problem is undecidable

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

- Integers, Reals etc. are finite
- References are infinite, but it is finite for every implementation
- Fundamental difference to *C-K*: reference parameters, local procedures, and procedure parameters that passes closures

☞ Half-band Turingmachine can be simulated on the runtime stack

$\Rightarrow$ Halting Problem is undecidable

Situation in other Programming Languages

Java, C#

- Set of references is potentially infinite

$\Rightarrow$ Infinite data types

$\Rightarrow$ Halting problem is undecidable

Java Bytecode, CLR

- References may be stored in operand stack
- Each entry in operand stacks can store machine words

$\Rightarrow$ Set of possible references is finite

$\Rightarrow$ No infinite data types

$\Rightarrow$ Situation analogous to *C-K*

$\Rightarrow$ Halting problem is decidable

Pascal, Ada, Modula-2

- Integers, Reals etc. are finite
- References are infinite, but it is finite for every implementation
- Fundamental difference to *C-K*: reference parameters, local procedures, and procedure parameters that passes closures

☞ Half-band Turingmachine can be simulated on the runtime stack

$\Rightarrow$ Halting Problem is undecidable

Conclusions

- The need of conservative program analyis approaches due to undecidability result is theoretically often wrong

☞ From a practical point it is often necessary due to state space explosion

- Software model checking (LTL, CTL) on pushdown systems is possible

⇒ This enables the possibility to experimentally compare the quality of conservative approximations.

Conclusions

- The need of conservative program analyis approaches due to undecidability result is theoretically often wrong

☞ From a practical point it is often necessary due to state space explosion

- Software model checking (LTL, CTL) on pushdown systems is possible

⇒ This enables the possibility to experimentally compare the quality of conservative approximations.

Conclusions

- The need of conservative program analyis approaches due to undecidability result is theoretically often wrong

☞ From a practical point it is often necessary due to state space explosion

- Software model checking (LTL, CTL) on pushdown systems is possible

⇒ This enables the possibility to experimentally compare the quality of conservative approximations.

Conclusions

- The need of conservative program analyis approaches due to undecidability result is theoretically often wrong

☞ From a practical point it is often necessary due to state space explosion

- Software model checking (LTL, CTL) on pushdown systems is possible

⇒ This enables the possibility to experimentally compare the quality of conservative approximations.